1. **What is Mongo DB?**

   It is Document Database/No SQL Database and it does not have fixed schema. We can store the unstructured data like audio and video etc. We can large amount data. Even though data represented in JSON but in MongoDB internally its stored as BSON format .

   JSON: Java Script Object Notation
   BSON: Binary JSON.

2. **Relational/SQL Vs Document/NoSQL Database?**

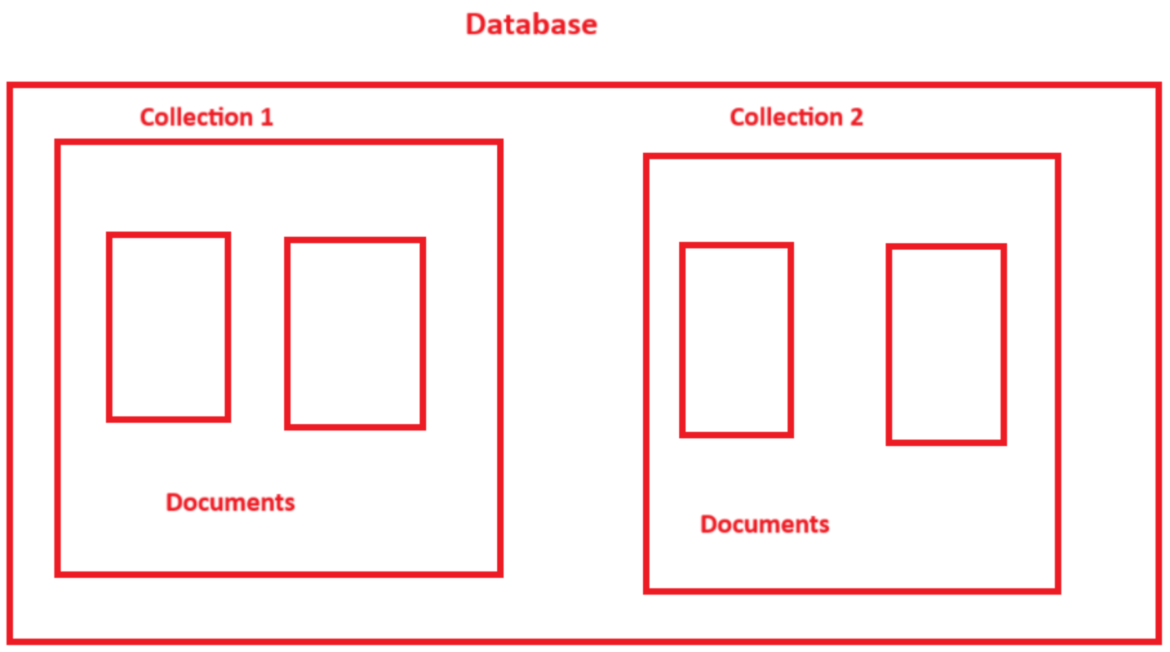
   Relational Database/SQL Database: Here data will be stored in tables and these tables have fixed schema. The data stored in tables have relationships like,

   One to one
   One to Many
   Many to Many
   Many to One

   To retrieve data from relational databases we have to write the join queries which collects the data from different tables.

   Ex: Oracle, MySQL etc.

   Document/NoSQL database: Here data will be stored in the form of document. Each document is independent of each other.



Internally MongoDB physical contains several logical databases.

Each data base have multiple collections. Collection is like table in relational database.
Each collection has multiple documents. Document is like record/row in relational database.

Ex: MongoDB, Cassendra, Amazon DynamoDB, Redis, HBase etc.

3. Key characteristics of MongoDB?

1. All information related to a document will be stored in single place. To retrieve data it is not required to perform join operation hence retrieval is very fast.
2. Documents are independent of each other with no schema. Hence we can store unstructured data like videos, audios etc.
3. We can perform operations like editing and deleting the existing document and inserting new document very easily.
4. As data stored in JSON which can be understandable by any programming language without any conversion.
5. We have store huge amount of data and hence scalability is more.

4. Explain Few commands of Mongo Shell?

Show Databases: To view the list of databases, use the command:

```
show dbs
```

Switch Database: You can switch to a specific database using the use command:

```
use mydatabase
```

Show Collections: To list all collections within the current database, use:

```
show collections
```

Insert Document: To insert a document into a collection, use the **insertOne()** or **insertMany()** methods.

exit: Allows us to exit the MongoDB shell session. We can also use **exit, exit(), or .exit** to quit the MongoDB shell.

To display the database you are using, type db:

```
db
```

The operation should return test, which is the default database.

To switch databases, issue the use <db> helper, as in the following example:

```
use <database>
```

To access a different database from the current database without switching your current database context, see the db.getSiblingDB() method.

'show databases'/'show dbs': Print a list of all available databases.

'show collections'/'show tables': Print a list of all collections for current database.

'show profile': Prints system.profile information.

'show users': Print a list of all users for current database.

'show roles': Print a list of all roles for current database.

'show log <type>': log for current connection, if type is not set uses 'global'

'show logs': Print all logs.

Change or Create a Database

You can change or create a new database by typing use then the name of the database.

```
use blog
```

Remember: In MongoDB, a database is not actually created until it gets content!

Create Collection using mongosh

```
db.createCollection("posts")
```

Remember: In MongoDB, a collection is not actually created until it gets content!

Insert Documents

There are 2 methods to insert documents into a MongoDB database.

insertOne()

To insert a single document, use the insertOne() method.

This method inserts a single object into the database.

```
db.posts.insertOne({
  title: "Post Title 1",
  body: "Body of post.",
  category: "News",
  likes: 1,
  tags: ["news", "events"],
  date: Date()
})
```

insertMany()

To insert multiple documents at once, use the insertMany() method.

This method inserts an array of objects into the database.

```
db.posts.insertMany([
  {
    title: "Post Title 2",
    body: "Body of post.",
    category: "Event",
    likes: 2,
    tags: ["news", "events"],
    date: Date()
  },
  {
    title: "Post Title 3",
    body: "Body of post.",
    category: "Technology",
    likes: 3,
    tags: ["news", "events"],
    date: Date()
  },
  {
    title: "Post Title 4",
    body: "Body of post.",
    category: "Event",
    likes: 4,
    tags: ["news", "events"],
    date: Date()
  }
])
```

Find Data

There are 2 methods to find and select data from a MongoDB
collection, find() and findOne().

find()

To select data from a collection in MongoDB, we can use the find() method.

This method accepts a query object. If left empty, all documents will be returned.

Example

```
db.posts.find( {category: "News"} )
```

findOne()

To select only one document, we can use the findOne() method.

This method accepts a query object. If left empty, it will return the first document it finds.

Note: This method only returns the first match it finds.

```
db.posts.findOne()
```

Projection: Both find methods accept a second parameter called projection. This parameter is an object that describes which fields to include in the results.

Update Document:

To update an existing document we can use **the updateOne() or updateMany()** methods.

The first parameter is a query object to define which document or documents should be updated.

The second parameter is an object defining the updated data.

updateOne()

The updateOne() method will update the first document that is found matching the provided query.

Let's see what the "like" count for the post with the title of "Post Title 1":

```
db.posts.find( { title: "Post Title 1" } )
```

Now let's update the "likes" on this post to 2. To do this, we need to use the $set operator.

```
db.posts.updateOne( { title: "Post Title 1" }, { $set: { likes: 2 } } )
```

Insert if not found

If you would like to insert the document if it is not found, you can use the **upsert** option.

Example

Update the document, but if not found insert it:

```
db.posts.updateOne(
  { title: "Post Title 5" },
  {
    $set:
      {
        title: "Post Title 5",
        body: "Body of post.",
        category: "Event",
        likes: 5,
        tags: ["news", "events"],
        date: Date()
      }
  },
  { upsert: true }
)
```

updateMany()

The updateMany() method will update all documents that match the provided query.

Example

Update likes on all documents by 1. For this we will use the $inc (increment) operator:

db.posts.updateMany({}, { $inc: { likes: 1 } })

Delete Documents

We can delete documents by using the methods **deleteOne() or deleteMany()**.

These methods accept a query object. The matching documents will be deleted.

deleteOne()

The deleteOne() method will delete the first document that matches the query provided.

Example

db.posts.deleteOne({ title: "Post Title 5" })

deleteMany()

The deleteMany() method will delete all documents that match the query provided.

Example

db.posts.deleteMany({ category: "Technology" })

MongoDB Query Operators

There are many query operators that can be used to compare and reference document fields.

Comparison

The following operators can be used in queries to compare values:

- $eq: Values are equal
- $ne: Values are not equal
- $gt: Value is greater than another value
- $gte: Value is greater than or equal to another value
- $lt: Value is less than another value
- $lte: Value is less than or equal to another value
- $in: Value is matched within an array

Logical

The following operators can logically compare multiple queries.

- $and: Returns documents where both queries match
- $or: Returns documents where either query matches
- $nor: Returns documents where both queries fail to match
- $not: Returns documents where the query does not match

Evaluation

The following operators assist in evaluating documents.

- $regex: Allows the use of regular expressions when evaluating field values
- $text: Performs a text search

- $where: Uses a JavaScript expression to match documents

MongoDB Update Operators

There are many update operators that can be used during document updates.

Fields

The following operators can be used to update fields:

- $currentDate: Sets the field value to the current date
- $inc: Increments the field value
- $rename: Renames the field
- $set: Sets the value of a field
- $unset: Removes the field from the document

Array

The following operators assist with updating arrays.

- $addToSet: Adds distinct elements to an array
- $pop: Removes the first or last element of an array
- $pull: Removes all elements from an array that match the query
- $push: Adds an element to an array

Aggregation Pipelines:

Aggregation operations allow you to group, sort, perform calculations, analyze data, and much more.

Aggregation pipelines can have one or more "stages". The order of these stages are important. Each stage acts upon the results of the previous stage.

$group : This aggregation stage groups documents by the unique _id expression provided.

```
db.listingsAndReviews.aggregate(
   [ { $group : { _id : "$property_type" } } ]
)
```

$limit: This aggregation stage limits the number of documents passed to the next stage.

```
db.movies.aggregate([ { $limit: 1 } ])
```

$project: This aggregation stage passes only the specified fields along to the next aggregation stage. This is the same projection that is used with the find() method.

```
db.restaurants.aggregate([
 {
   $project: {
     "name": 1,
     "cuisine": 1,
     "address": 1
   }
 },
 {
   $limit: 5
 }
])
```

This will return the documents but only include the specified fields.

$sort: This aggregation stage groups sorts all documents in the specified sort order.

```
db.listingsAndReviews.aggregate([
 {
   $sort: { "accommodates": -1 }
 },
 {
   $project: {
     "name": 1,
     "accommodates": 1
   }
 },
 {
   $limit: 5
 }
])
```

This will return the documents sorted in descending order by the accommodates field. The sort order can be chosen by using 1 or -1. 1 is ascending and -1 is descending.

$match: This aggregation stage behaves like a find. It will filter documents that match the query provided. Using $match early in the pipeline can improve performance since it limits the number of documents the next stages must process.

```
db.listingsAndReviews.aggregate([
 { $match : { property_type : "House" } },
 { $limit: 2 },
 { $project: {
   "name": 1,
   "bedrooms": 1,
   "price": 1
 }}
])
```

$addFields: This aggregation stage adds new fields to documents.

```
db.restaurants.aggregate([
 {
   $addFields: {
     avgGrade: { $avg: "$grades.score" }
   }
 },
 {
   $project: {
     "name": 1,
     "avgGrade": 1
   }
 },
 {
   $limit: 5
 }
])
```

This will return the documents along with a new field, avgGrade, which will contain the average of each restaurants grades.score.

$count: This aggregation stage counts the total amount of documents passed from the previous stage.

```
db.restaurants.aggregate([
  {
    $match: { "cuisine": "Chinese" }
  },
  {
    $count: "totalChinese"
  }
])
```

This will return the number of documents at the $count stage as a field called "totalChinese".

$lookup: This aggregation stage performs a left outer join to a collection in the same database.

There are four required fields:

- **from**: The collection to use for lookup in the same database
- **localField**: The field in the primary collection that can be used as a unique identifier in the from collection.
- **foreignField**: The field in the from collection that can be used as a unique identifier in the primary collection.
- **as**: The name of the new field that will contain the matching documents from the from collection.

Example

In this example, we are using the "sample_mflix" database loaded from our sample data in the Intro to Aggregations section.

```
db.comments.aggregate([
  {
    $lookup: {
      from: "movies",
      localField: "movie_id",
      foreignField: "_id",
      as: "movie_details",
    },
  },
  {
    $limit: 1
  }
])
```

$out: This aggregation stage writes the returned documents from the aggregation pipeline to a collection. The $out stage must be the last stage of the aggregation pipeline.

```
db.listingsAndReviews.aggregate([
  {
    $group: {
      _id: "$property_type",
      properties: {
        $push: {
          name: "$name",
          accommodates: "$accommodates",
          price: "$price",
        },
      },
    },
  },
  { $out: "properties_by_type" },
])
```

The first stage will group properties by the property_type and include the name, accommodates, and price fields for each. The $out stage will create a new collection called properties_by_type in the current database and write the resulting documents into that collection.

Schema Validation:

By default MongoDB has a flexible schema. This means that there is no strict schema validation set up initially.

Schema validation rules can be created in order to ensure that all documents in a collection share a similar structure.

MongoDB supports JSON Schema validation. The $jsonSchema operator allows us to define our document structure.

```
db.createCollection("posts", {
 validator: {
  $jsonSchema: {
   bsonType: "object",
   required: [ "title", "body" ],
   properties: {
    title: {
     bsonType: "string",
     description: "Title of post - Required."
    },
    body: {
     bsonType: "string",
     description: "Body of post - Required."
    },
    category: {
     bsonType: "string",
     description: "Category of post - Optional."
    },
    likes: {
     bsonType: "int",
     description: "Post like count. Must be an integer - Optional."
    },
    tags: {
     bsonType: ["string"],
     description: "Must be an array of strings - Optional."
```

```
        },
        date: {
          bsonType: "date",
          description: "Must be a date - Optional."
        }
      }
    }
  }
})
```

This will create the posts collection in the current database and specify the JSON Schema validation requirements for the collection.

MongoDB Data API:

The MongoDB Data API can be used to query and update data in a MongoDB database without the need for language specific drivers.

Language drivers should be used when possible, but the MongoDB Data API comes in handy when drivers are not available or drivers are overkill for the application.

Read & Write with the MongoDB Data API

The MongoDB Data API is a pre-configured set of HTTPS endpoints that can be used to read and write data to a MongoDB Atlas database.

With the MongoDB Data API, you can create, read, update, delete, or aggregate documents in a MongoDB Atlas database.

Cluster Configuration

In order to use the Data API, you must first enable the functionality from the Atlas UI.

From the MongoDB Atlas dashboard, navigate to **Data API** in the left menu.

Select the data source(s) you would like to enable the API on and click **Enable the Data API**.

Access Level

By default, no access is granted. Select the access level you'd like to grant the Data API. The choices are: No Access, Read Only, Read and Write, or Custom Access.

Data API Key

In order to authenticate with the Data API, you must first create a Data API key.

Click **Create API Key**, enter a name for the key, then click **Generate API Key**.

Be sure to copy the API key and save it somewhere safe. *You will not get another chance to see this key again.*

Data API Endpoints

In the previous example, we used the findOne endpoint in our URL.

There are several endpoints available for use with the Data API.

All endpoints start with the Base URL: https://data.mongodb-api.com/app/<Data API App ID>/endpoint/data/v1/action/

MongoDB Drivers

The MongoDB Shell (mongosh) is great, but generally you will need to use MongoDB in your application. To do this, MongoDB has many language drivers.

The language drivers allow you to interact with your MongoDB database using the methods you've learned so far in `mongosh` but directly in your application.

These are the current officially supported drivers:

- C
- C++
- C#
- Go
- Java
- Node.js
- PHP
- Python
- Ruby
- Rust
- Scala
- Swift

Indexes

Indexes are data structures that make it simple to navigate across the collection's data set. They help to execute queries and find documents that match the query criteria without a collection scan.

These are the following different types of indexes in MongoDB:

Single field

MongoDB can traverse the indexes either in the **ascending** or **descending order** for single-field index

```
db.students.createIndex({“item”:1})
```

In this example, we are creating a single index on the item field and 1 here represents the filed is in ascending order.

A **compound index** in MongoDB contains multiple single filed indexes separated by a comma. MongoDB restricts the number of fields in a compound index to a maximum of 31.

```
db.students.createIndex({“item”: 1, “stock”:1})
```

Here, we create a compound index on item: 1, stock:1